Mathematical Modeling Project Report

Arnav Subedi, Noah Horne, Dastan Askarbekov, Iskandar Akhundov

14.05.2024

Dr. Ivan Ovsyannikov

Dzmitry Rumiantsau

We hereby declare that we are the sole authors of this project report and have not used any sources other than those listed in the bibliography and identified as references throughout the report.

_____ _____ _____ _____

Contents

1 The Damped Harmonic Oscillator **2**

1.1 Initial Value Problem for the Damped Harmonic Oscillation 3

1.1.1 The Explicit Euler's Method 4

1.1.2 The Implicit Euler's Method 7

1.1.3 Python's Built-In Solver, scipy.integrate.solve_ivp 11

1.1.4 Error Analysis 12

1.2 Boundary Value Problem for the Damped Harmonic Oscillation 14

1.2.1 Shooting Method: 14

1.2.2 Finite Difference Method: 16

1.2.3 Python's Built-In Solver, scipy.integrate.solve_bvp 18

2 The Heat Equation **19**

3 Appendix **21**

3.1 The Damped Harmonic Oscillator 21

3.1.1 Code for numerical analysis using different methods for IVP 21

3.1.2 Code for error analysis for Damped Oscillation (IVP) 24

3.1.3 Code for numerical analysis of shooting method for BVP 25

3.1.4 Code for numerical analysis of finite difference method for BVP 26

3.1.5 Code for numerical analysis of Scipy's solve_BVP method for BVP . . . 27

3.1.6 Code for simulation of heat flow in 2d system 28

References **32**

Abstract

In this report, we study the numerical methods to study Ordinary Differential Equations (ODE) and Partial Differential Equations (PDE). In this particular project, we were tasked to study the damped harmonic oscillator, $\ddot{x} + 2\zeta\dot{x} + x = 0$, and the heat equation, $\frac{\partial u}{\partial t} = \alpha\nabla^2 u$. The numerical approximations of the solutions to the ODE and PDE were carried out using Python3 and different methods of numerical solutions were studied and utilized (Explicit and Implicit Euler's method for Initial Value Problems (IVP); Shooting method and finite difference method for Boundary Valued Function (BVP); built-in python functions form scipy package). The approximated solutions were compared with the analytical solutions (if possible) and validated. The numerical approximation and visualization were done by both authors and the codes by both are attached to the .zip file.

1 The Damped Harmonic Oscillator

The Ordinary Differential Equation modelling a Damped Harmonic Oscillator is given by,

$$\ddot{x} + 2\zeta\dot{x} + x = 0 \tag{1}$$

where ζ is the damping coefficient. Notice that 1 is a second-order linear differential equation for which the analytical solution can be easily solved. First, one can convert the following second-order ODE into the system of two first-order ODE by letting $\dot{x} = v \Rightarrow \ddot{x} = \dot{v}$.

$$\begin{aligned} \ddot{x} &= -(2\zeta\dot{x} + x) = \dot{v} \\ \dot{x} &= v \end{aligned}$$

This simplifies to a system of ODEs that depends on $x, v, \dot{x}$ and $\dot{v}$:

$$\begin{aligned} \dot{x} &= v \\ \dot{v} &= -(2\zeta v + x) \end{aligned} \tag{2}$$

Notice that the eigenvalues of the given differential equation are given by $\lambda_{1,2} = -\zeta \pm \sqrt{\zeta^2 - 1}$. Thus, the general solution of the given ODE is given by:

$$x(t) = \alpha e^{(-\zeta + \sqrt{\zeta^2 - 1})t} + \beta e^{(-\zeta - \sqrt{\zeta^2 - 1})t} \tag{3}$$

for some α, β determined by the initial values or boundary conditions of the ODE. It is also important to note the first derivative of $x(t)$, $\dot{x}(t)$, as it will be relevant during the study of IVP.

$$\dot{x}(t) = \alpha \cdot (-\zeta + \sqrt{\zeta^2 - 1}) \cdot e^{(-\zeta + \sqrt{\zeta^2 - 1})t} + \beta \cdot (-\zeta - \sqrt{\zeta^2 - 1}) \cdot e^{(-\zeta - \sqrt{\zeta^2 - 1})t} \tag{4}$$

An important observation one can make is that if $|\zeta| < 1$, the eigenvalues are complex–thus one must utilize Euler's equation of exponential functions with complex exponents to get the analytical solutions–which is therefore:

$$x(t) = e^{-\zeta t}(\gamma \cos{(\sqrt{|\zeta^2 - 1|}t)} + \delta \sin{(\sqrt{|\zeta^2 - 1|}t)}) \tag{5}$$

for some γ, β. The solution will still be valid in the exponential form, but it's easier to study in trigonometric form. However, this form is only valid for $|\zeta| < 1$.

1.1 Initial Value Problem for the Damped Harmonic Oscillation

Provided the initial conditions for an Ordinary Differential Equation (ODE), one can precisely determine the solution of a differential equation, and consequently model its behavior. By extrapolating the behavior of the solution from the initial value, the entire solution to the ODE can be approximated to a high degree of accuracy, forming an at times exact, or close-to-exact solution. When considering the solution of an n-th order differential equation, one must be provided with n initial conditions corresponding to each total derivative of the function being examined. For a differential equation of type $f(y^{(n)}(x), y^{(n-1)}(x), \cdots, y'(x), y(x), g(x)) = 0$, the values of $y(x_0), y'(x_0), \cdots y^{(n)}(x_0)$ specify the initial value problem of the ODE. This differential equation problem is known as the Initial Value Problem (IVP), and is a strikingly recurrent problem seen across a plethora of real world applications.

In the task assigned, the initial value problem concerns the solution to the trajectory of a second order damped harmonic oscillator ODE. The damped harmonic oscillation was given initial conditions of $x(0) = 1$ and $\dot{x}(0) = 0$. Using this initial condition, the analytic solution of 1 was determined (see above calculations) as:

$$x(t) = \left(\frac{\zeta + \sqrt{\zeta^2 - 1}}{2\sqrt{\zeta^2 - 1}}\right) \cdot e^{(-\zeta + \sqrt{\zeta^2 - 1})t} + \left(\frac{-\zeta + \sqrt{\zeta^2 - 1}}{2\sqrt{\zeta^2 - 1}}\right) \cdot e^{(-\zeta - \sqrt{\zeta^2 - 1})t} \tag{6}$$

There are three cases one needs to keep in mind while evaluating the analytical solution, complex roots, real roots and repeated roots. The full solution is given by:

$$x(t) = \begin{cases} \left(\frac{\zeta + \sqrt{\zeta^2 - 1}}{2\sqrt{\zeta^2 - 1}}\right) \cdot e^{(-\zeta + \sqrt{\zeta^2 - 1})t} + \left(\frac{-\zeta + \sqrt{\zeta^2 - 1}}{2\sqrt{\zeta^2 - 1}}\right) \cdot e^{(-\zeta - \sqrt{\zeta^2 - 1})t}, & \zeta^2 > 1 \\ (\zeta t + 1) \cdot e^{-\zeta t}, & \zeta^2 = 1 \\ e^{-\zeta t}\left(\cos{(\sqrt{|\zeta^2 - 1|}t)} + \frac{\zeta}{\sqrt{|\zeta^2 - 1|}} \cdot \sin{(\sqrt{|\zeta^2 - 1|}t)}\right), & \zeta^2 < 1 \end{cases} \tag{7}$$

To model the given ODE, three methods and tool kits were utilized–the explicit Euler's method, the implicit Euler's method and Python's built-in ODE solver, scipy.integrate.solve_ivp.

1.1.1 The Explicit Euler's Method

Let us first consider the explicit Euler's method, a popularly used incremental approach which discretely extrapolates the solution of an ODE from a given initial value. Given a first-order differential equation of the form,

$$y'(t) = f(t, y(t)),\ y(t_0) = y_0,\ f : [t_0, \infty) \times \mathbb{R}^d \to \mathbb{R}^d \tag{8}$$

one can attempt to approximate the solution by exploiting the step-ratio limit definition of the derivative. From first principles, note that the first-order linear approximation of a function is given as:

$$y'(t) \approx \frac{y(t+h) - y(t)}{h} \Rightarrow y(t+h) \approx y(t) + h \cdot f(t, y(t)) \tag{9}$$

Using this first-order linear approximation, one can simply make the jump to a recursive formulation of the approximation, as seen below.

$$y_{n+1} = y_n + h \cdot f(t_n, y_n) \text{ where } t_n = t_0 + nh \tag{10}$$

This recursive formulation allows a discretely defined extrapolation from an initial value that linearly approximates the function at a given point, assuming that we know the value of the function at that point. By continuously applying this algorithm for a given step size h, one can create a large dataset (1D numerical grid) of these consecutively generated points that discretely describe the solution of the ODE. As one may infer, this recursive formulation can be implemented iteratively in Python code to provide an accurate approximation for the solution of an ODE, as seen below.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from scipy.optimize import fsolve

def damped_harmonic_oscillator(t, y, zeta):
    x1, x2 = y
    dx1dt = x2
    dx2dt = -2 * zeta * x2 - x1
    return [dx1dt, dx2dt]

def explicit_euler(f, t0, y0, t_end, h, zeta):
    t_values = np.arange(t0, t_end, h)
```

```
    y_values = np.zeros((len(t_values), len(y0)))
    y_values[0] = y0

    for i in range(1, len(t_values)):
        y_values[i] = y_values[i-1] + h * np.array(f(t_values[i-1],
    y_values[i-1], zeta))

    return t_values, y_values
```

The Explicit Euler's Method is a 'First-Order Accurate' numerical approximation method, meaning it is only serviceable for first-order ODEs. This is due to its underlying reliance on the definition of the first derivative and linear approximations as defined above. Moreover, for ODEs of the first order, it is only accurate for small step sizes, as larger step sizes compound error in a pseudo-exponential fashion–a larger deviation from the original function will lead to a more deviant consecutive approximation, and so on. This deviation is clearly evident in figure 11, 12 and 13. As one can see, the error in the approximation grows in an almost exponential fashion as the step size that we use increases. Due to the first-order stability of the explicit Euler numerical approximation method, it is deemed as an unstable numerical approximation method. This is also a direct result of the finite radius of convergence for the step size, the interval of step sizes such that the approximation does not diverge or fluctuate in a chaotic manner. Later on, the specifics of this error accumulation and chaotic behaviour of divergent approximations will be discussed in section 1.1.4.

Now, let us apply this method to the ODE in question, 1. One can take t as the numerical grid for the approximation with the incremental step size h, which will be varied and examined at runtime. We know that the recursive equation is solvable as the initial condition is specified. The figure below denotes the result of the python code above used to calculate the solution to 1 using the Explicit Euler's Method. Note that the program is run for 3 distinct values of ζ–0.25, 1 and 2. It is important to note that the error analyzed in these figures is not against the time step h, but rather how the error changes with respect to time. This is done to directly observe the effects of error accumulation over time, and the step size divergence will be demonstrated and discussed in section 1.1.4, respectively.

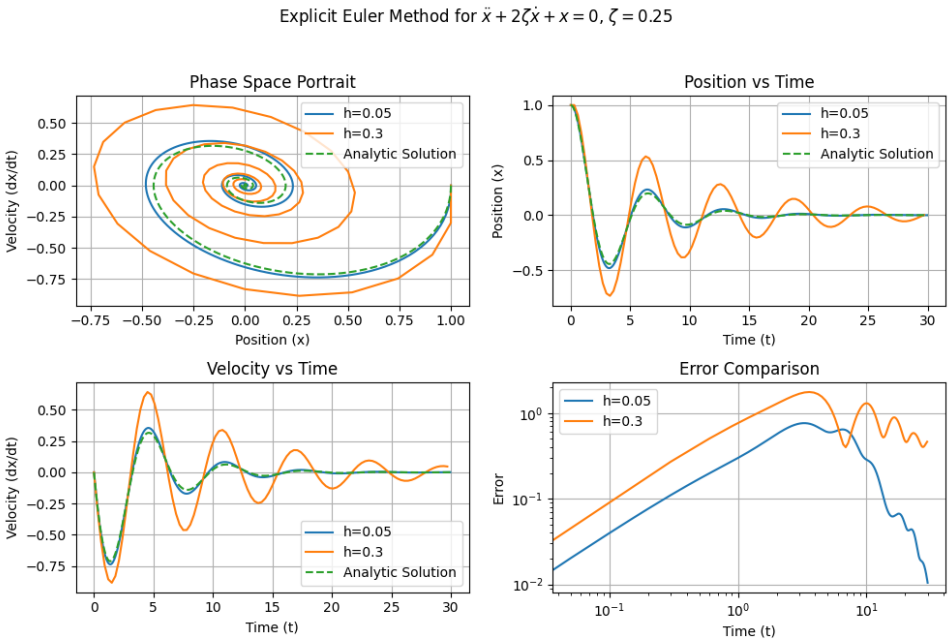

Figure 1: Plots of $\ddot{x} + 2\zeta\dot{x} + x = 0, \zeta = 0.25$ with Explicit Euler Method

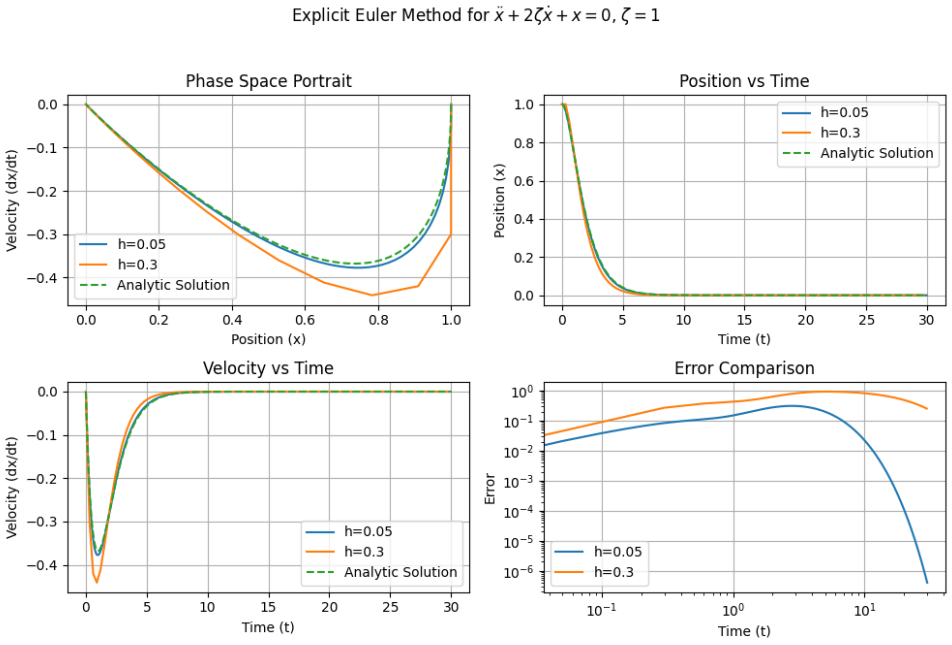

Figure 2: Plots of $\ddot{x} + 2\zeta\dot{x} + x = 0, \zeta = 1$ with Explicit Euler Method

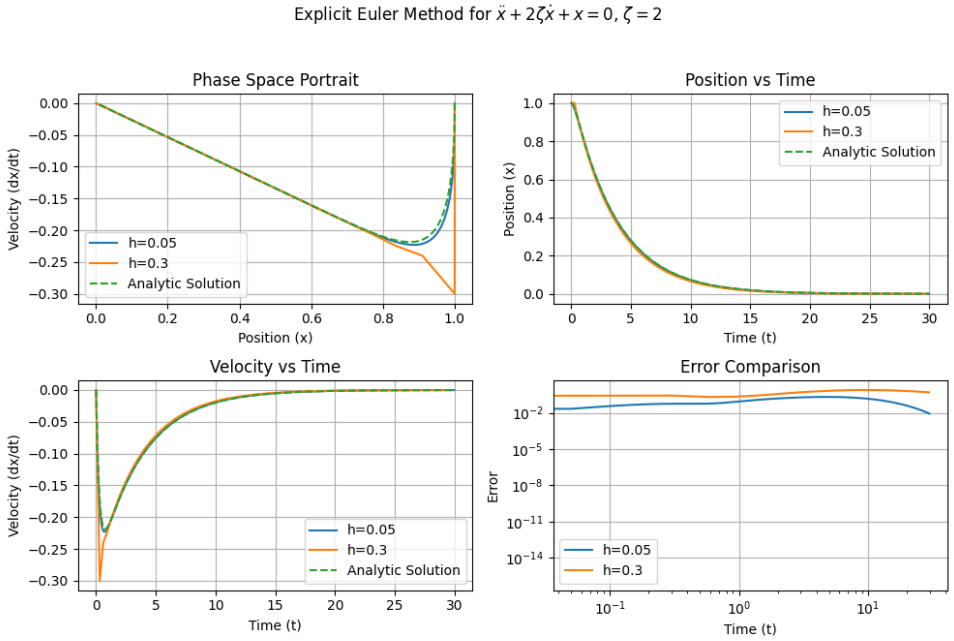

Figure 3: Plots of $\ddot{x} + 2\zeta\dot{x} + x = 0, \zeta = 2$ with Explicit Euler Method

As one can observe, in the figures below denoting the error of the method with respect to time, the error resulting from the smaller time step is also smaller, demonstrating this relationship visually. This is clear as less deviation can be compounded with a smaller time step as the solution continuously remains close to the analytical solution for a smaller time step, as observed in both the phase space diagram and the two position and velocity plots.

1.1.2 The Implicit Euler's Method

Alternatively to the explicit Euler's method, which forward iterates from an initial value as a means of solving the ODE, the implicit Euler's Method (The Backwards Euler Method), is a similar numerical approximation method to Euler's method in that it still is of order one in time. The reason the Backwards Euler Method is also called an implicit method is due to the way that we define the relationship between consequent approximations in our solution. The Backwards Euler's Method functions based on the following implicitly recursive relation:

$$y_{k+1} = y_k + hf(t_{k+1}, y_{k+1}) \tag{11}$$

Note that this method only slightly differs in that $f(t_k, y_k)$ is replaced by the implicit function with respect to $y_k + 1$, $f(t_{k+1}, y_{k+1})$, and it is derived as follows, by integrating an arbitrary function of t and y, $f(t, y(t))$ (Butcher, 2016).

$$\frac{dy}{dt} = f(t, y),$$

where we integrate from t_n to $t_{n+1} = t_n + h$, giving us,

$$y(t_{n+1}) - y(t_n) = \int_{t_n}^{t_{n+1}} f(t, y(t))dt \tag{12}$$

Now, for small enough h, we can approximate the area under the curve as just the rectangle of the area such that the height is the value of the function at t_{n+1}.

$$\begin{aligned} \implies y(t_{n+1}) - y(t_n) &\approx h \cdot f(t_{n+1}, y(t_{n+1})) \\ y_{k+1} &= y_k + h \cdot f(t_{k+1}, y_{k+1}) \end{aligned} \tag{13}$$

One can consequently formulate this method in Python code in an iterative manner, looping over a time grid given an initial condition and repeatedly solving f, as seen below.

```
def implicit_euler(f, t0, y0, t_end, h, zeta):
    def implicit_step(y_next, t, y_current, h, zeta):
        return y_next - y_current - h * np.array(f(t + h, y_next, zeta))

    t_values = np.arange(t0, t_end, h)
    y_values = np.zeros((len(t_values), len(y0)))
    y_values[0] = y0

    for i in range(1, len(t_values)):
        y_next_guess = y_values[i-1]
        y_values[i] = fsolve(implicit_step, y_next_guess, args=(t_values
    [i-1], y_values[i-1], h, zeta))

    return t_values, y_values
```

Note that similar to the explicit Euler's method, this method also has a finite radius of convergence for the step size of its approximation, however the convergence is inverted. This means the approximation method will only converge to the appropriate solution with a step size outside of the radius, h. This region is the interval $h \in (0, 1]$ (Butcher, 2016). In the figure below, this radius of convergence is demonstrated and displayed as a plot on the complex plane for all step sizes that converge for the implicit Euler method. In fact, "The region for a discrete stable system by Backward Euler Method is a circle with radius 0.5 which is located at (0.5, 0) in the z-plane" (Chen, 2018) Notice that this statement implies that the region of convergence takes the form of the complement of an offset complex disk, where values outside of the disk are more stable. This idea will come into play later in section 1.1.4, as the error with respect to the time step appears in a downward parabolic shape centred around 0.5. peaking at the centre of where the instability disk would be.

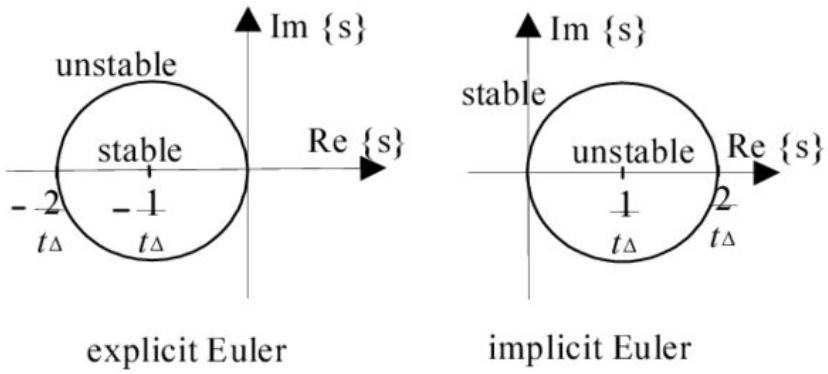

Figure 4: Disk of convergence for the Explicit and Implicit Euler Method on the Complex Plane (Niiranen, 1999)

Now that we have established the relevant theory, we will examine the results of this method's implementation in Python code as written above for the appropriate values of ζ, 0.25, 1 and 2 respectively. Again, it is important to note that the error in the bottom left is not the error plotted with respect to the time step, as the error propagating with respect to time is another component of interest, and can be appropriately examined as done for the Explicit Euler Method as well.

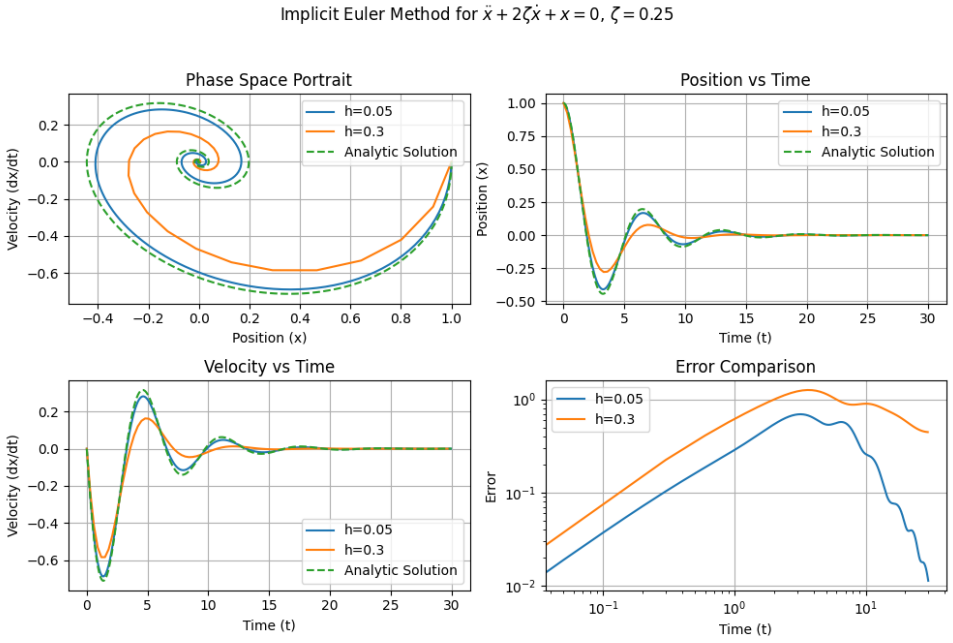

Figure 5: Plots of $\ddot{x} + 2\zeta\dot{x} + x = 0, \zeta = 0.25$ with Implicit Euler Method

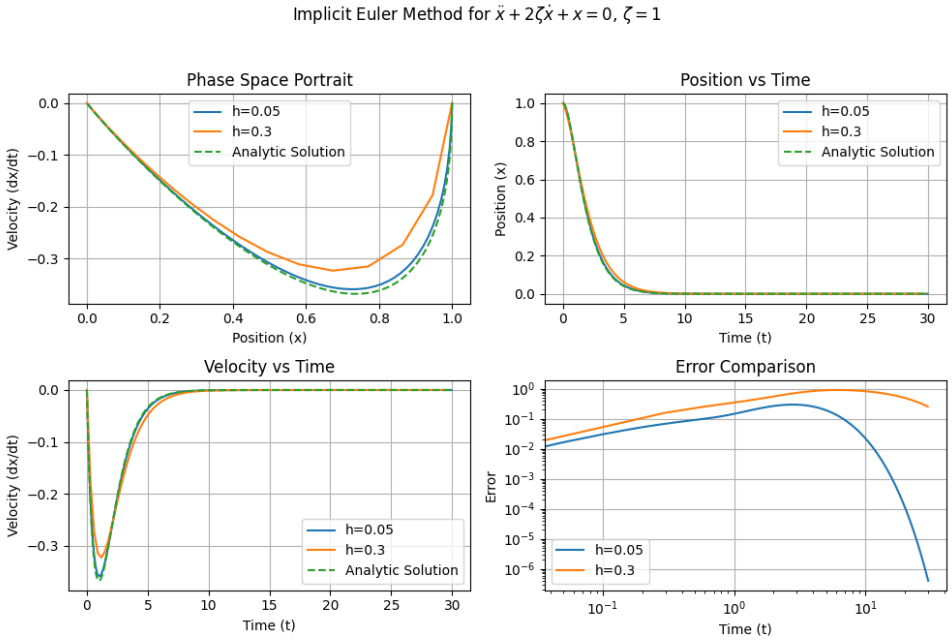

Figure 6: Plots of $\ddot{x} + 2\zeta\dot{x} + x = 0, \zeta = 1$ with Implicit Euler Method

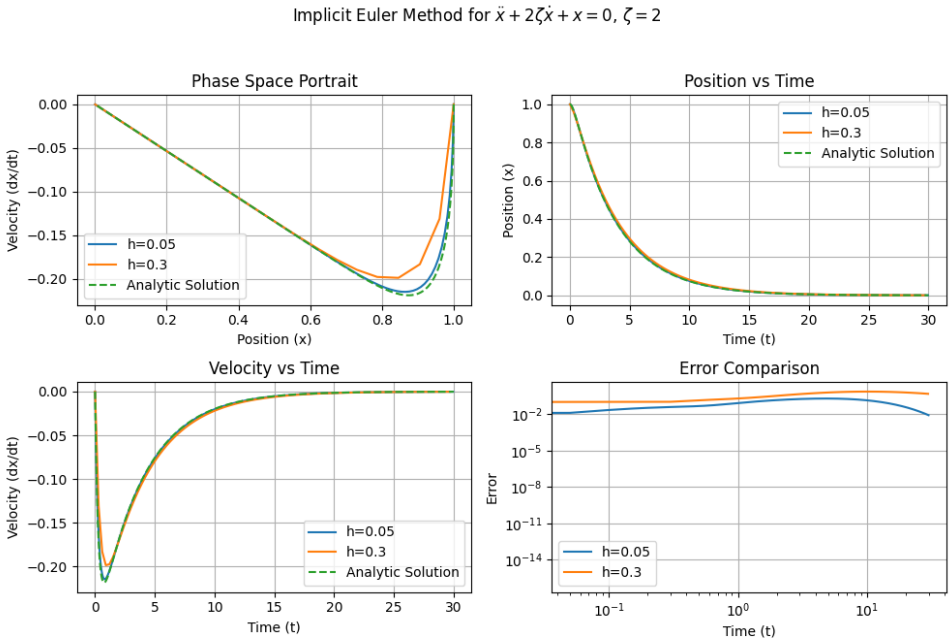

Figure 7: Plots of $\ddot{x} + 2\zeta\dot{x} + x = 0, \zeta = 2$ with Implicit Euler Method

Notice that in the figures for the error for each value of ζ, the error with respect to time exhibits very similar behavior to that observed for the Explicit Euler Method. It is also clear that the error for position with a smaller time step $h = 0.05$ is consistently smaller than the error with the larger time step $h = 0.3$

1.1.3 Python's Built-In Solver, scipy.integrate.solve_ivp

As stated in the documentation of the SciPy library, the *scipy.integrate.solve_ivp* function is a numerical approximation function which, "numerically integrates a system of ordinary differential equations given an initial value" **(docs.scipy.org)**. While the function itself has multiple methods that it uses to perform the solution of ODEs, its default explicit method is called 'RK45'. This acronym is short for the **Explicit Runge-Kutta** method of order 5(4). Unlike the extremely low accuracy of the explicit and implicit Euler's Method, which only had an order of 1, this default method is a fifth-order accurate formula. It functions by performing a quartic interpolation that fills in the gaps between the explicitly determined approximated points, creating a globally dense output, which gives much more information than a method such as the explicit Euler's method, which has a very fixed step size and leads to a non-dense solution. It is important to note that while the Euler method is a Runge-Kutta method, the method that the builtin function uses belongs to a family of more accurate solvers which us temporal discretization to approximate the solution of such ODEs. Below is a simple implementation of the built in solver:

```
1 def solve_using_scipy(t0, y0, t_end, zeta, h):
2     t_eval = np.arange(t0, t_end, h)
3     solution = solve_ivp(damped_harmonic_oscillator, (t0, t_end), y0,
    args=(zeta,), t_eval=t_eval)
4     return solution.t, solution.y.T
```

Further below are the results of the implementation of the built-in solver on the ODE. It is clear from the diagram as well that the error this method yields is extremely small and almost negligible due to the method's high level of accuracy.

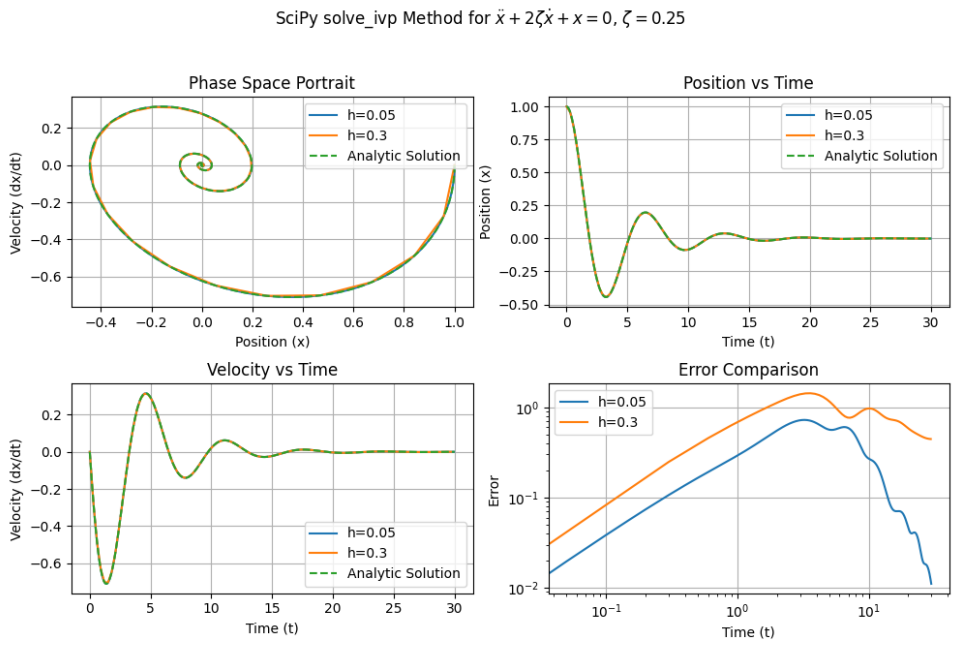

Figure 8: Plots of $\ddot{x} + 2\zeta\dot{x} + x = 0, \zeta = 0.25$ with Scipy's solve_IVP Method

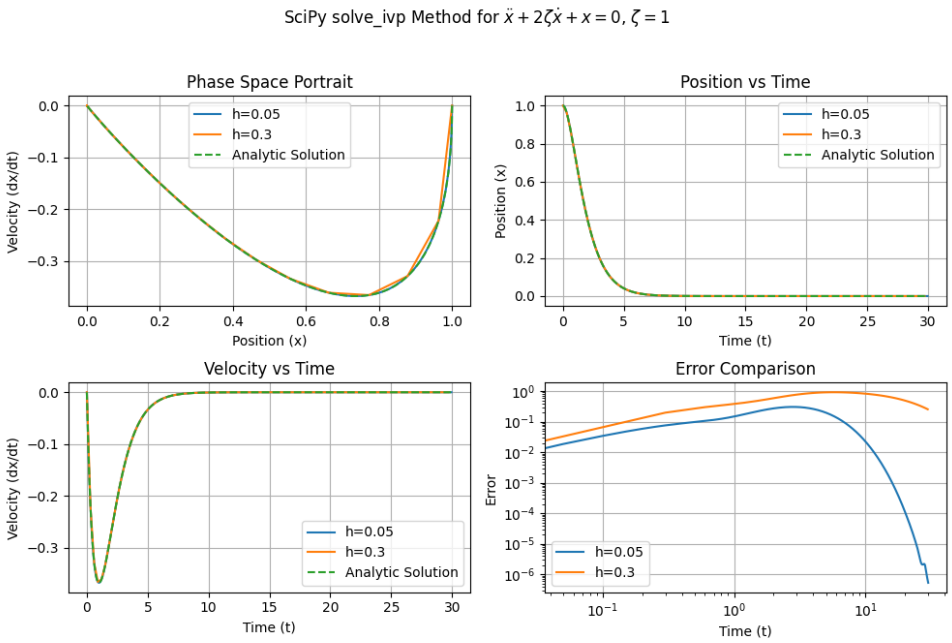

Figure 9: Plots of $\ddot{x} + 2\zeta\dot{x} + x = 0, \zeta = 1$ with Scipy's solve_IVP Method

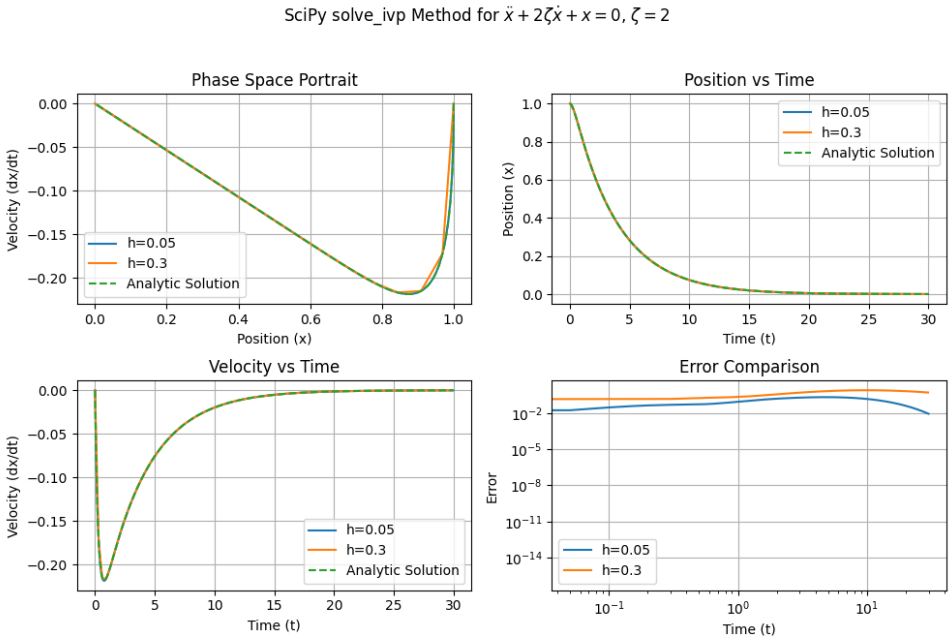

Figure 10: Plots of $\ddot{x} + 2\zeta\dot{x} + x = 0, \zeta = 2$ with Scipy's solve_IVP Method

1.1.4 Error Analysis

As briefly aforementioned, the error of these methods varies significantly in form due to the variety in their approach to the initial value problem. While the explicit Euler's method takes on a more bottom up approach, forward iterating to provide an accurate and precise approximation, the latter methods work through backwards iteration and further interpolation which yields

drastically different error results for a varying step size. As discussed in their appropriate sections, the convergence regions for the explicit and implicit Euler methods are practically inverted from one another, all convergent solutions determined by the boundary or inverted boundary of a complex disk. As for the Runge-Kutta and its 5(4) order accuracy it is clear that it will behave differently due to its temporal discretization and quartic interpolation. Below are the plots of each methods error ran at 100 different step sizes ranging form 0.01 to 1 time unit over a range of 30 time units. Moreover, the three plots correspond to the three different values for ζ which were examined for the system, 0.25, 1 and 2 respectively.

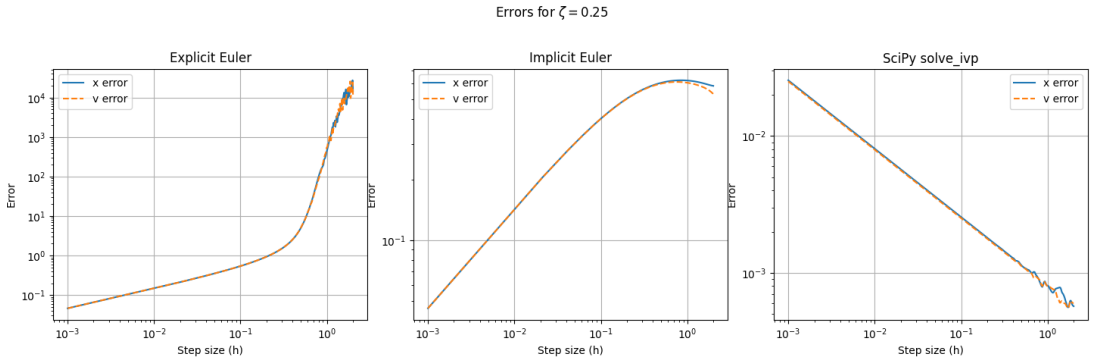

Figure 11: Error for $\zeta = 0.25$ for each method

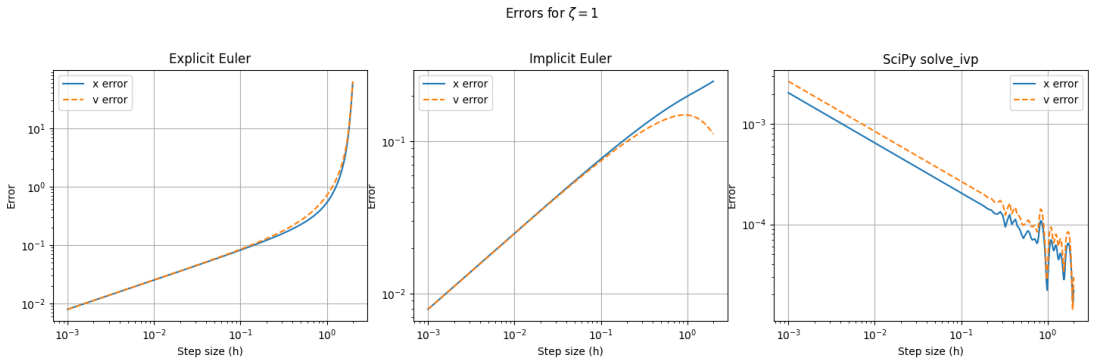

Figure 12: Error for $\zeta = 1$ for each method

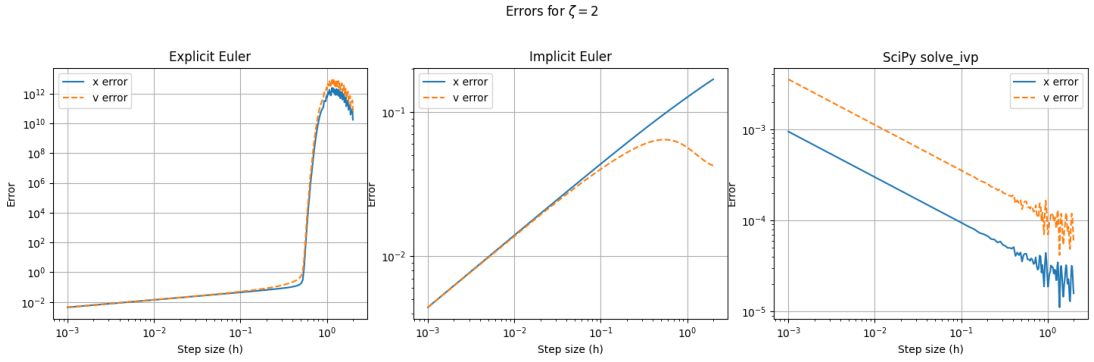

Figure 13: Error for $\zeta = 2$ for each method

Let's examine the error for each of the methods: For the error in the explicit Euler's method, it is clearly growing in a pseudo-exponential fashion for small damping coefficients, most likely due to how the larger the step becomes, the larger the accumulation of error is on average across the runtime of the method. To elaborate, when the linear approximation of the function given an initial value is calculated, it slightly deviates from the expected value. While the initial value is exact to the analytical solution, after the first step the solution becomes approximate, as it is no longer based on the solution itself. Rather, it is extrapolated a small amount by a linear approximation. Although this step is small, it implies that in consecutive iterative steps, the linear approximation begins with a value that is deviant from the real solution by a small margin. Consequently, these deviations compound on one another, and it is what we see in the figures above.

As for the implicit Euler's method, one can describe the concave-down parabolic shape of the error very well based off of the shape of its complex domain of convergence, as denoted in 4. Clearly, the domain of stable convergence for the implicit Euler method lies outside of a disk centered around 0.5 with a radius of 0.5. As seen in the plots, which are in a log log scale, the concave-down parabolic shape has a maximum very close to 0.5, showing that the region of least stability is at the center of the circle, which is directly observed in all three plots for the velocity.

1.2 Boundary Value Problem for the Damped Harmonic Oscillation

In this section one will consider the boundary value problem for the same differential equation, 1, only this time with respect to the consideration of a boundary value problem. Below are the methods used in numerically approximating the solution of the harmonic oscillator ODE through the application of two distinct boundary-value solving methods–the Shooting Method, and the Finite Difference Method. Descriptions of these methods are denoted below, alongside their associated implementations in the Python3 language.

Specifically, the boundary conditions that are to be considered when using these methods are given as follows for a parameter value of $\zeta = 0.25$:

$$x(0) = 1, x(9) = 0, t \in [0, 9]$$

Thus, let's look at each method individually.

1.2.1 Shooting Method:

Consider the following equation:

$$y''(t) = f(t, y(t), y'(t)),\ y(t_0) = y_0,\ y(t_1) = y_1,\ f : [t_0, t_1] \times \mathbb{R} \times \mathbb{R} \to \mathbb{R} \tag{14}$$

The shooting method works by converting the given BVP into IVP by converting one

of the boundary conditions into an initial condition. For example, we can choose $y(t_0) = y_0$ into an initial condition by guessing an initial value for $y'(t_0)$. This transforms the BVP into IVP. Now, with the guessed initial value, one can integrate the ODE numerically from t_0 to t_1 using a numerical ODE solver such as Runge-Kutta or Euler Method. After the integration, we check if the solution satisfies the remaining boundary condition, $y(t_1) = y_1$. If it doesn't, we adjust the initial guess for $y'(a)$ and repeat all the steps again. We do this by checking the roots of the equation and then adjusting the $y'(t_0)$ to get the desired second boundary value as root.

The shooting method was implemented in Python and was used to numerically approximate the Boundary Value Problem of interest. Below is the python implementation of the shooting method:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from scipy.optimize import fsolve

zeta = 0.25
x0 = 1
tf = 9

def f(t, r):
    x, v = r
    dxdt = v
    dvdt = -(2 * zeta * v + x)
    return dxdt, dvdt

@np.vectorize
def xf(v0):
    sol = solve_ivp(f, (0, tf), (x0, v0))
    x, v = sol.y
    return x[-1]

v0 = fsolve(xf, 2)[0]

t = np.linspace(0, tf, 100)

sol = solve_ivp(f, (0, tf), (x0, v0), t_eval= t)
x, v = sol.y

plt.figure(figsize=(8,4))
plt.plot(t, x, label = "position")
plt.plot(t, v, label = "Velocity")
plt.scatter([0, 9], [1, 0])
plt.xlabel('time (s)')
plt.ylabel('amplitude')
plt.title(fr'Shooting Method at $\zeta = {zeta}$')
```

```
plt.grid()
plt.legend()
plt.show()
```

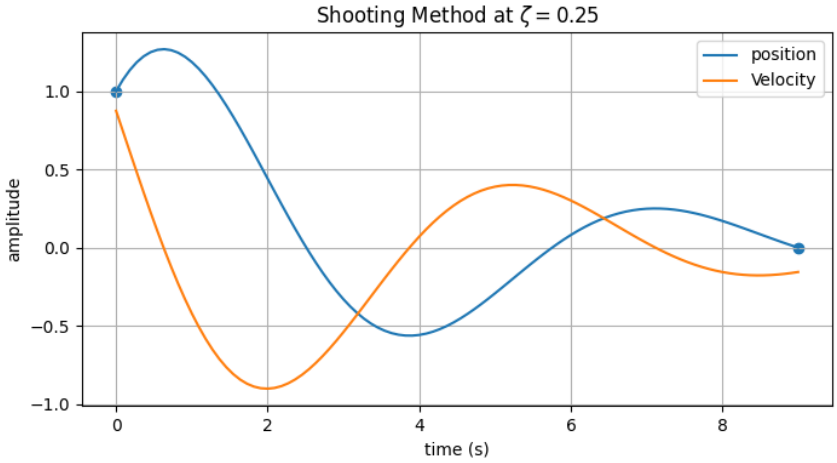

Figure 14: Plots of position and velocity for $\zeta = 0.25$ using Shooting Method

The result so obtained using the shooting method perfectly matches the expectation and the analytic solution for the given boundary value and ζ parameter.

1.2.2 Finite Difference Method:

The same ODE can be solved by considering the discretization of the domain and the ODE. The finite Difference Method works by first dividing the interval $[a, b]$ into n equal subintervals of length h such that $x_0 = a$ and $x_n = b$. h is given by $\frac{b-a}{n}$. Then, the ODE is approximated at each grid point using the finite difference approximations, such that

$$\frac{dy}{dx} = \frac{y_{i+1} - y_{i-1}}{2h}$$

$$\frac{d^2y}{dx^2} = \frac{y_{i-1} - 2y_i + y_{i+1}}{h^2}$$

This leads to a system of $n + 1$ algebraic equations that can be used to solve for y_i and numerically approximate y for all the grid points. The discretized ODE and boundary condition normally lead to the system of equations, which can be represented in matrix form and this system can be solved using a standard system of equation solvers like Gaussian elimination or LU decomposition. This gives us the value of y_i, $\forall i$ thus giving us the numerical solution.

The finite difference method was implemented in Python and was used to numerically approximate the Boundary Value Problem of interest. Below is the Python implementation of the finite difference method:

```
import numpy as np
import matplotlib.pyplot as plt
```

```
3
4  n = 100
5  h = 9 / n
6  zeta = 0.25
7  A = np.zeros((n+1, n+1))
8  A[0, 0] = 1
9  A[n, n] = 1
10 for i in range(1, n):
11     A[i, i-1] = 1-zeta*h
12     A[i, i] = -2 + h**2
13     A[i, i+1] = 1 + zeta * h
14
15 b = np.ones(n+1)
16 b[1:-1] = 0
17 b[-1] = 0
18
19 y = np.linalg.solve(A, b)
20
21 v = np.zeros_like(y)
22 v[0] = (y[1] - y[0]) / h
23 v[1:-1] = (y[2:] - y[:-2]) / (2 * h)
24 v[-1] = (y[-1] - y[-2]) / h
25
26 t = np.linspace(0, 9, n+1)
27
28 plt.figure(figsize=(8,4))
29 plt.plot(t, y, label = "position")
30 plt.plot(t, v, label = "velocity")
31 plt.scatter([0, 9], [1, 0])
32 plt.xlabel('time (s)')
33 plt.ylabel('amplitude')
34 plt.title(fr'Finite Difference Method at $\zeta = {zeta}$')
35 plt.legend()
36 plt.grid()
37 plt.show()
```

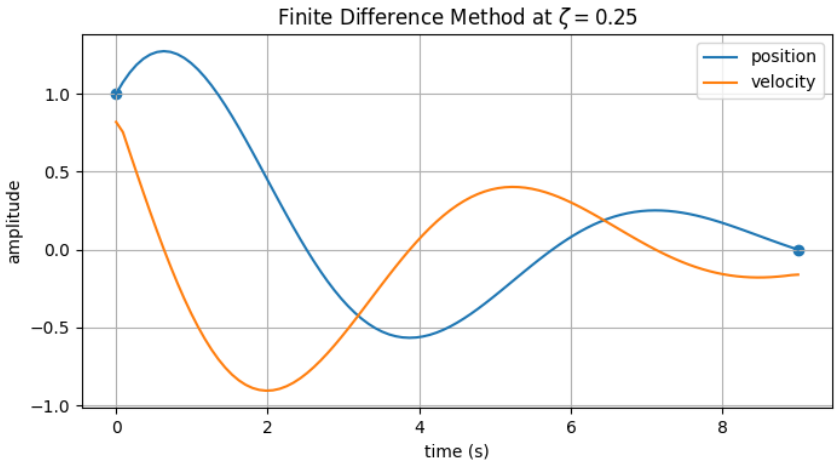

Figure 15: Plots of position and velocity for $\zeta = 0.25$ using Finite Difference Method

1.2.3 Python's Built-In Solver, scipy.integrate.solve_bvp

As stated in the documentation of the SciPy library, the scipy_integrate.solve_bvp function is a numerical approximation function which, "numerically solves a first order system of ODEs subject to two-point boundary conditions" (docs.scipy.org). The method used is based on a collocation approach (which involves approximating the solution of the differential equation by a piecewise polynomial) combined with the finite difference method discussed above. The collocation method transforms the BVP into a system of non-linear equations which is solved iteratively using Newton's method. The accuracy of solve_bvp varies depending on the degree of the polymonial (4th to 6th or even higher). The method uses adaptive mesh refinement to improve accuracy by concentrating computational efforts in regions of high error. This method generally achieves higher order of accuracy than the finite difference method and the shooting method discussed previously. Although the finite difference method can achieve 4th order of accuracy, the solve_bvp method is still more efficient, as it can use less grid points to achieve higher order of accuracy through its adaptive strategy. Below is the code that implements solve_bvp.

```
import numpy as np
import matplotlib.pyplot as plt

n = 100
h = 9 / n
zeta = 0.25
A = np.zeros((n+1, n+1))
A[0, 0] = 1
A[n, n] = 1
for i in range(1, n):
    A[i, i-1] = 1-zeta*h
    A[i, i] = -2 + h**2
    A[i, i+1] = 1 + zeta * h

```

```
b = np.ones(n+1)
b[1:-1] = 0
b[-1] = 0

y = np.linalg.solve(A, b)

v = np.zeros_like(y)
v[0] = (y[1] - y[0]) / h
v[1:-1] = (y[2:] - y[:-2]) / (2 * h)
v[-1] = (y[-1] - y[-2]) / h

t = np.linspace(0, 9, n+1)

plt.figure(figsize=(8,4))
plt.plot(t, y, label = "position")
plt.plot(t, v, label = "velocity")
plt.scatter([0, 9], [1, 0])
plt.xlabel('time (s)')
plt.ylabel('amplitude')
plt.title(fr'Finite Difference Method at $\zeta = {zeta}$')
plt.legend()
plt.grid()
plt.show()
```

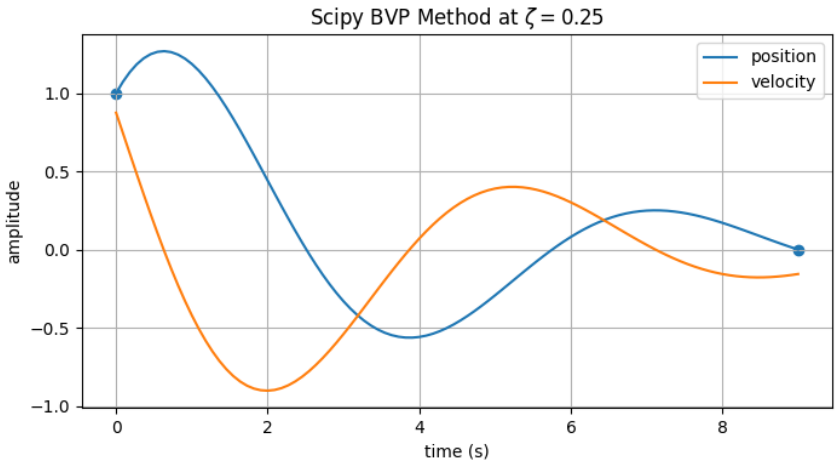

Figure 16: Plots of position and velocity for $\zeta = 0.25$ using Scipy's solve_BVP Method

2 The Heat Equation

One can study the temporal variation of heat in 2d in an object using the following Partial Differential Equation (PDE)

$$\frac{\partial u(t,x,y)}{\partial t} = \alpha \left(\frac{\partial^2 u(t,x,y)}{\partial x^2} + \frac{\partial^2 u(x,y,t)}{\partial y^2} \right) \tag{15}$$

The following PDE can also be numerically approximated using the same finite difference method as stated previously. One can construct a similar system of equations using the discretization of the derivatives giving rise to a $n+1$ system of equations that can be solved. For our particular study, the following conditions were applied:

Over a surface of a rectangle $[0, 100] \times [0, 100]$, with $\alpha = 3$ The system was studied and simulated with the following boundary conditions:

$$u(t,x,y) = \begin{cases} 0, & t = 0,\ 0 < y < 100,\ 0 < x < 100 \\ 0, & x = 0,\ 0 < y < 100 \\ 0, & x = 100,\ 0 < y < 100, \\ 200, & 0 \leq x \leq 100,\ y = 0 \\ 200, & 0 \leq x \leq 100,\ y = 100 \end{cases} \tag{16}$$

The Python code solves the system of equations and thus gets the value of $u(t, x, y)$ at a particular time, t. This is then plotted as a heatmap on a 2d plane and the simulation runs over time demonstrating the heat flow. In the study, two codes were developed and both of the codes are listed in the Appendix.

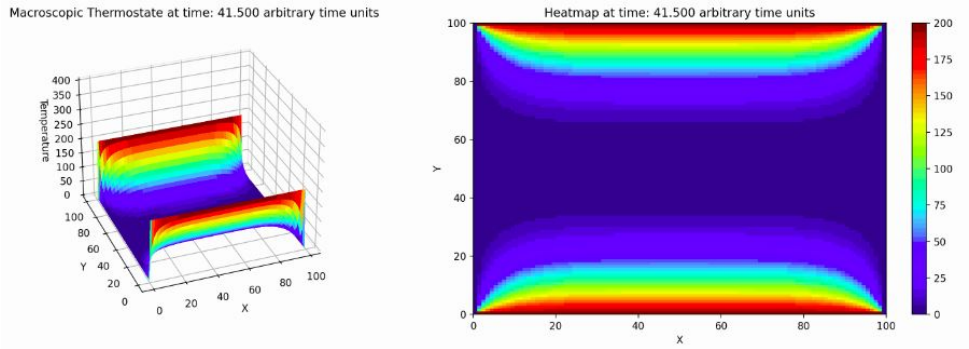

Figure 17: 3d distribution of heat and 2d state of heat at $t = 41.500$ units

3 Appendix

3.1 The Damped Harmonic Oscillator

3.1.1 Code for numerical analysis using different methods for IVP

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from scipy.optimize import fsolve

def damped_harmonic_oscillator(t, y, zeta):
    x1, x2 = y
    dx1dt = x2
    dx2dt = -2 * zeta * x2 - x1
    return [dx1dt, dx2dt]

def explicit_euler(f, t0, y0, t_end, h, zeta):
    t_values = np.arange(t0, t_end, h)
    y_values = np.zeros((len(t_values), len(y0)))
    y_values[0] = y0

    for i in range(1, len(t_values)):
        y_values[i] = y_values[i-1] + h * np.array(f(t_values[i-1],
    y_values[i-1], zeta))

    return t_values, y_values

def implicit_euler(f, t0, y0, t_end, h, zeta):
    def implicit_step(y_next, t, y_current, h, zeta):
        return y_next - y_current - h * np.array(f(t + h, y_next, zeta))

    t_values = np.arange(t0, t_end, h)
    y_values = np.zeros((len(t_values), len(y0)))
    y_values[0] = y0

    for i in range(1, len(t_values)):
        y_next_guess = y_values[i-1]
        y_values[i] = fsolve(implicit_step, y_next_guess, args=(t_values
    [i-1], y_values[i-1], h, zeta))

    return t_values, y_values

def solve_using_scipy(t0, y0, t_end, zeta, h):
    t_eval = np.arange(t0, t_end, h)
    solution = solve_ivp(damped_harmonic_oscillator, (t0, t_end), y0,
    args=(zeta,), t_eval=t_eval)
```

```
    return solution.t, solution.y.T

def analytical_solution(t, zeta):
    if zeta**2 < 1:
        A = np.sqrt(np.abs(zeta**2 - 1))
        x = np.exp(-zeta*t) * (np.cos(A*t) + ((zeta)/(A))*np.sin(A*t))
        v = np.exp(-zeta*t) * (np.sin(A*t)*(-(zeta**2)/(A) - A))
    elif zeta**2 == 1:
        x = np.exp(-zeta*t) * (zeta*t + 1)
        v = np.exp(-zeta*t) * (-t*zeta**2)
    elif zeta**2 > 1:
        A = (zeta + np.sqrt(zeta**2 - 1))/(2*np.sqrt(zeta**2 - 1))
        B = (-zeta + np.sqrt(zeta**2 - 1))/(2*np.sqrt(zeta**2 - 1))
        C = -zeta + np.sqrt(zeta**2 - 1)
        D = -zeta - np.sqrt(zeta**2 - 1)
        x = A*np.exp(C*t) + B*np.exp(D*t)
        v = A*C*np.exp(C*t) + B*D*np.exp(D*t)
    return x, v


def compute_error(analytic_solution, numerical_solution):
    return np.linalg.norm(analytic_solution - numerical_solution, axis
    =1)

def plot_results(method_name, zeta, t_values, y_values, t_analytic,
    y_analytic, errors, step_sizes):
    fig, axs = plt.subplots(2, 2, figsize=(10, 7))
    fig.suptitle(f'{method_name} Method for $\\ddot{{x}} + 2\\zeta\\dot
    {{x}} + x = 0$, $\\zeta = {zeta}$')
    axs[0, 0].plot(y_values[0][:, 0], y_values[0][:, 1], label=f'h={
    step_sizes[0]}')
    axs[0, 0].plot(y_values[1][:, 0], y_values[1][:, 1], label=f'h={
    step_sizes[1]}')
    axs[0, 0].plot(y_analytic[:, 0], y_analytic[:, 1], label='Analytic
    Solution', linestyle='--')
    axs[0, 0].set_title('Phase Space Portrait')
    axs[0, 0].set_xlabel('Position (x)')
    axs[0, 0].set_ylabel('Velocity (dx/dt)')
    axs[0, 0].legend()
    axs[0, 0].grid(True)

    axs[0, 1].plot(t_values[0], y_values[0][:, 0], label=f'h={step_sizes
    [0]}')
    axs[0, 1].plot(t_values[1], y_values[1][:, 0], label=f'h={step_sizes
    [1]}')
    axs[0, 1].plot(t_analytic, y_analytic[:, 0], label='Analytic
    Solution', linestyle='--')
```

```
    axs[0, 1].set_title('Position vs Time')
    axs[0, 1].set_xlabel('Time (t)')
    axs[0, 1].set_ylabel('Position (x)')
    axs[0, 1].legend()
    axs[0, 1].grid(True)

    axs[1, 0].plot(t_values[0], y_values[0][:, 1], label=f'h={step_sizes[0]}')
    axs[1, 0].plot(t_values[1], y_values[1][:, 1], label=f'h={step_sizes[1]}')
    axs[1, 0].plot(t_analytic, y_analytic[:, 1], label='Analytic Solution', linestyle='--')
    axs[1, 0].set_title('Velocity vs Time')
    axs[1, 0].set_xlabel('Time (t)')
    axs[1, 0].set_ylabel('Velocity (dx/dt)')
    axs[1, 0].legend()
    axs[1, 0].grid(True)

    axs[1, 1].loglog(t_values[0], errors[0], label=f'h={step_sizes[0]}')
    axs[1, 1].loglog(t_values[1], errors[1], label=f'h={step_sizes[1]}')
    axs[1, 1].set_title('Error Comparison')
    axs[1, 1].set_xlabel('Time (t)')
    axs[1, 1].set_ylabel('Error')
    axs[1, 1].legend()
    axs[1, 1].grid(True)

    plt.tight_layout(rect=[0, 0.03, 1, 0.95])
    plt.show()

initial_conditions = [1, 0]
t0 = 0
t_end = 30
zetas = [0, 0.25, 1, 2]
step_sizes = [0.05, 0.3]

for zeta in zetas:
    t_analytic = np.linspace(t0, t_end, 1000)
    x_analytic, v_analytic = analytical_solution(t_analytic, zeta)
    y_analytic = np.vstack((x_analytic, v_analytic)).T

    for method in ["Explicit Euler", "Implicit Euler", "SciPy solve_ivp"]:
        if method == "Explicit Euler":
            t_values = []
            y_values = []
            errors = []
            for h in step_sizes:
```

```
                t, y = explicit_euler(damped_harmonic_oscillator, t0, initial_conditions, t_end, h, zeta)
                t_values.append(t)
                y_values.append(y)
                errors.append(compute_error(y_analytic[:len(y)], y))
        elif method == "Implicit Euler":
            t_values = []
            y_values = []
            errors = []
            for h in step_sizes:
                t, y = implicit_euler(damped_harmonic_oscillator, t0, initial_conditions, t_end, h, zeta)
                t_values.append(t)
                y_values.append(y)
                errors.append(compute_error(y_analytic[:len(y)], y))
        else:
            t_values = []
            y_values = []
            errors = []
            for h in step_sizes:
                t, y = solve_using_scipy(t0, initial_conditions, t_end, zeta, h)
                t_values.append(t)
                y_values.append(y)
                errors.append(compute_error(y_analytic[:len(y)], y))
        plot_results(method, zeta, t_values, y_values, t_analytic, y_analytic, errors, step_sizes)
```

3.1.2 Code for error analysis for Damped Oscillation (IVP)

```
from IVP_Analysis_diff_methods import *
import matplotlib.pyplot as plt

def compute_error_(h, t0, t_end, zeta, method):
    t = np.arange(t0, t_end, h)
    x_analytic, v_analytic = analytical_solution(t, zeta)
    y_analytic = np.vstack((x_analytic, v_analytic)).T

    if method == explicit_euler or method == implicit_euler:
        t, y = method(damped_harmonic_oscillator, t0, initial_conditions, t_end, h, zeta)
    elif method == solve_using_scipy:
        t, y = method(t0, initial_conditions, t_end, zeta, h)

    matching_indices = np.intersect1d(np.arange(len(y_analytic)), np.arange(len(y)))
    x_error = np.linalg.norm(y_analytic[matching_indices, 0] - y[matching_indices, 0])
```

```
    v_error = np.linalg.norm(y_analytic[matching_indices, 1] - y[
    matching_indices, 1])

    return x_error, v_error

initial_conditions = [1, 0]
t0 = 0
t_end = 30
zetas = [0, 0.25, 1, 2]
step_sizes = np.linspace(0.001, 2, 100)
methods = [explicit_euler, implicit_euler, solve_using_scipy]
method_names = ["Explicit Euler", "Implicit Euler", "SciPy solve_ivp"]

for zeta in zetas:
    fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 5))
    fig.suptitle(f'Errors for $\\zeta = {zeta}$')

    axes = [ax1, ax2, ax3]
    for ax, method, method_name in zip(axes, methods, method_names):
        x_errors = []
        v_errors = []
        for h in step_sizes:
            x_error, v_error = compute_error_(h, t0, t_end, zeta, method
    )
            x_errors.append(x_error)
            v_errors.append(v_error)

        ax.loglog(step_sizes, x_errors, label=f'x error')
        ax.loglog(step_sizes, v_errors, label=f'v error', linestyle='--'
    )

        ax.set_title(f"{method_name}")
        ax.set_xlabel('Step size (h)')
        ax.set_ylabel('Error')
        ax.legend()
        ax.grid(True)

    plt.tight_layout(rect=[0, 0.03, 1, 0.95])
    plt.show()
```

3.1.3 Code for numerical analysis of shooting method for BVP

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
from scipy.optimize import fsolve

zeta = 0.25
```

```
x0 = 1
tf = 9

def f(t, r):
    x, v = r
    dxdt = v
    dvdt = -(2 * zeta * v + x)
    return dxdt, dvdt

@np.vectorize
def xf(v0):
    sol = solve_ivp(f, (0, tf), (x0, v0))
    x, v = sol.y
    return x[-1]

v0 = fsolve(xf, 2)[0]

t = np.linspace(0, tf, 100)

sol = solve_ivp(f, (0, tf), (x0, v0), t_eval= t)
x, v = sol.y

plt.figure(figsize=(8,4))
plt.plot(t, x, label = "position")
plt.plot(t, v, label = "Velocity")
plt.scatter([0, 9], [1, 0])
plt.xlabel('time (s)')
plt.ylabel('amplitude')
plt.title(fr'Shooting Method at $\zeta = {zeta}$')
plt.grid()
plt.legend()
plt.show()
```

3.1.4 Code for numerical analysis of finite difference method for BVP

```
import numpy as np
import matplotlib.pyplot as plt

n = 100
h = 9 / n
zeta = 0.25
A = np.zeros((n+1, n+1))
A[0, 0] = 1
A[n, n] = 1
for i in range(1, n):
    A[i, i-1] = 1-zeta*h
    A[i, i] = -2 + h**2
    A[i, i+1] = 1 + zeta * h
```

```
14
15 b = np.ones(n+1)
16 b[1:-1] = 0
17 b[-1] = 0
18
19 y = np.linalg.solve(A, b)
20
21 v = np.zeros_like(y)
22 v[0] = (y[1] - y[0]) / h
23 v[1:-1] = (y[2:] - y[:-2]) / (2 * h)
24 v[-1] = (y[-1] - y[-2]) / h
25
26 t = np.linspace(0, 9, n+1)
27
28 plt.figure(figsize=(8,4))
29 plt.plot(t, y, label = "position")
30 plt.plot(t, v, label = "velocity")
31 plt.scatter([0, 9], [1, 0])
32 plt.xlabel('time (s)')
33 plt.ylabel('amplitude')
34 plt.title(fr'Finite Difference Method at $\zeta = {zeta}$')
35 plt.legend()
36 plt.grid()
37 plt.show()
```

3.1.5 Code for numerical analysis of Scipy's solve_BVP method for BVP

```
1 from scipy.integrate import solve_bvp
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 zeta = 0.25
6
7 def ode(t, y):
8     return np.array([y[1], -2 * zeta * y[1] - y[0]])
9
10 def bc(ya, yb):
11     return np.array([ya[0] - 1, yb[0]])
12
13 t = np.linspace(0, 9, 100)
14 y = np.zeros((2, t.size))
15 y[0, 0] = 1
16
17 sol = solve_bvp(ode, bc, t, y)
18
19 plt.figure(figsize=(8, 4))
20 plt.plot(sol.x, sol.y[0], label="position")
21 plt.plot(sol.x, sol.y[1], label="velocity")
```

```
plt.scatter([0, 9], [1, 0])
plt.xlabel('time (s)')
plt.ylabel('amplitude')
plt.title(fr'Scipy BVP Method at $\zeta = {zeta}$')
plt.legend()
plt.grid()
plt.show()
```

3.1.6 Code for simulation of heat flow in 2d system

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.animation as animation

LENGTH = 100
max_iter_time = 500

alpha = 3
d_step = 1

# Calculated Parameters from Fixed Params
t_step = (d_step ** 2)/(4 * alpha)
gamma = (alpha * t_step) / (d_step ** 2)

# Now we create our Mesh
u = np.empty((max_iter_time, LENGTH, LENGTH))

# Create initial condition
u_initial = 0
u.fill(u_initial)

# Boundary Conditions
u_top = 200
u_left = 0
u_right = 0
u_bottom = 200

# Set Constant Boundary Conditions
u[:, (LENGTH-1):, :] = u_top
u[:, :, :1] = u_left
u[:, :1, 1:] = u_bottom
u[:, :, (LENGTH-1):] = u_right

def finite_diff_calculate(u):
    for k in range(0, max_iter_time-1, 1):
        for i in range(1, LENGTH-1, d_step):
            for j in range(1, LENGTH-1, d_step):
```

```
                u[k+1, i, j] = gamma * (u[k, i+1, j] + u[k, i-1, j] + u[k, i, j+1] + u[k, i, j-1] - 4*u[k, i, j]) + u[k, i, j]
    return u

# Calculate and reassign all thermostate values
u = finite_diff_calculate(u)


def plot_render(uk, k, fig):
    ax1 = fig.add_subplot(121, projection='3d')
    ax2 = fig.add_subplot(122)

    # Plot 3D Projection
    ax1.set_title(f"Macroscopic Thermostate at time: {k*t_step:.3f} arbitrary time units")
    ax1.set_xlabel("X")
    ax1.set_ylabel("Y")
    ax1.set_zlabel("Temperature")
    x = np.linspace(0, LENGTH-1, LENGTH)
    y = np.linspace(0, LENGTH-1, LENGTH)
    X, Y = np.meshgrid(x, y)
    ax1.axes.set_zlim3d(bottom=0,top=400)
    ax1.plot_surface(X, Y, uk, cmap=plt.cm.jet, linewidth=0, antialiased=False)
    ax1.view_init(elev=30, azim=k*0.5)  # Adjust azim multiplier for rotation speed


    # Plot Heatmap
    ax2.set_title(f"Heatmap at time: {k*t_step:.3f} arbitrary time units")
    ax2.set_xlabel("X")
    ax2.set_ylabel("Y")
    heatmap = ax2.pcolormesh(uk, cmap=plt.cm.jet, vmin=0, vmax=200)
    plt.colorbar(heatmap, ax=ax2)
    fig.subplots_adjust(wspace=1, hspace=5)


    plt.tight_layout()

    return fig


def animation_controller(k, fig):
    fig.clear()
    plot_render(u[k], k, fig)

```

```
fig = plt.figure(figsize=(15, 5))
ani = animation.FuncAnimation(fig, animation_controller, fargs=(fig,),
    interval=1, repeat=False, frames=max_iter_time)
ani.save("Heat_Eq_solution.gif", writer='pillow')
```

Alternative solution for bonus problem:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

plate_length = 101
max_iter_time = 300

alpha = 3
delta_x = 1

delta_t = (delta_x ** 2) / (4 * alpha)
gamma = (alpha * delta_t) / (delta_x ** 2)

u = np.empty((max_iter_time, plate_length, plate_length))

u_initial = 0

u_top = 200.0
u_bottom = 200.0

u.fill(u_initial)

u[:, (plate_length - 1):, :] = u_top
u[:, 0, :] = u_bottom

def calculate(u):
    for k in range(0, max_iter_time - 1):
        for i in range(1, plate_length - 1):
            for j in range(1, plate_length - 1):
                u[k + 1, i, j] = gamma * (u[k][i + 1][j] + u[k][i - 1][j
    ] + u[k][i][j + 1] + u[k][i][j - 1] - 4 * u[k][i][j]) + u[k][i][j]

    return u

def plotheatmap(u_k, k):
    plt.clf()
    plt.title(f"Temperature at t = {k * delta_t:.3f} unit time")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.pcolormesh(u_k, cmap=plt.cm.jet, vmin=0, vmax=200)
```

```
    plt.colorbar()
    return plt

u = calculate(u)

fig = plt.figure()
def animate(k):
    plotheatmap(u[k], k)

ani = animation.FuncAnimation(fig, animate, interval=10, frames=
    max_iter_time, repeat=False)
plt.show()
```

References

Butcher, J. C. (2016). *Numerical methods for ordinary differential equations*. John Wiley & Sons.

Chen, W.-K. (2018). *The circuits and filters handbook (five volume slipcase set)*. CRC Press.

Niiranen, J. (1999). Fast and accurate symmetric euler algorithm for electromechanical simulations. In *Electrimacs 99 (modelling and simulation of electric machines converters an& systems)* (pp. I–71).